SACO 2009 Day 1 Solutions

SACO Scientific Committee

1 Artifact Pieces

Once you realise that the two pieces of the artefact can be treated independently, the problem is very similar to the Guessing Game practice problem. If you are looking for the western piece, simply treat `E` and `B` the same. It makes no difference whether you seek the western or eastern piece.

Unlike the Guessing Game, the limited power of the metal detector means that you must use a binary search.

1.1 Binary search

A binary search is for finding an item with very few queries, where we can tell whether a query is too large or too small. It keeps track of the low and high bounds of where the item could be (in this case 0 and 1 000 000), and repeatedly querying the middle of the range. If the middle is too low, we replace the low bound with the middle, and if it is too high replace the high bound with the middle. Thus the sought item is always between the two bounds, and when the bounds meet we have found the item.

2 Guardian

This problem is reducible to a well known NP-Complete problem called set cover. This means there is no "fast" solution for it. Any of the below algorithms can be used, even in combination.

2.1 Brute force

This solution maps all possible knight placements around a UFO, and minimizing the number of knights that do get placed. This has a complexity of $O(N \times 8^{2N})$ for the input test data which is too much for pure brute force. However, realising that the limited range of the knight you can apply this algorithm to small subsets of the data. This should get you 2 test cases. In fact, it would be easier to do case 1 and 2 by hand.

2.2 Investigate Sparse Sets

In some cases the UFOs are clustered into small patterns that are scattered at huge intervals. Using a union-find for elements that are roughly less than 100 squares away, you can isolate these patterns so that you can solve them individually, since the patterns repeat you can find the optimal solutions for the sparse cases using this method.

2.3 Greedy Removal

This solution places knights on every position that can guard a UFO. The knights are then ranked in in terms of how many UFOs each knight guards. Also rate the UFOs in terms of how many knights guard them. The knights are then tested to see whether they are nessesary, starting with the lowest scoring knights. Only if a knight is deemed necessary does it get confirmed. The benefit of this solution is that you can much easier to spot where you should use a few lower ranked knights instead of a higher ranked knight. This can cause you to add more knights than is necessary.

2.4 Greedy Add

This is similar to Greedy Removal, except that it confirms the knights with a higher score rather than remove those with the lowest scores.

2.5 Random-Tiebreaker

In the event that more than one knight has the same score. You can randomly select which knight to add or remove (depending on your algorithm), since at this point it is very difficult to tell which knights will give a more optimal score. Therefore, you can run your program over and over again, trying to decreasethe number of knights you place.

3 Stacker

The way to approach this problem is with a technique called dynamic programming. The recurrence relation can be derived as follows. First we assume that we have a function **best_score**. This function returns the best possible score we can get if we where to start the game when there where only i values left on the stack. This is under the assumption that both players are playing optimally (which is what we want). Then the answer is $\mathbf{best\_score}(N)$. So lets define it!

If $i \leq 0$ then we can't pop any values off the stack, so the best score we can hope for is 0.

Now notice that if the best score for the opposing player for i is x, then our score is $\mathbf{sum}(i) - x$. (Where $\mathbf{sum}(i)$ is the sum of the first i numbers). Using this fact we can define **best_score** for $i > 0$ as follows:

$$\mathbf{best\_score}(i) = \mathbf{sum}(i) - \mathbf{min}_{1 \leq j \leq M}\{\mathbf{best\_score}(i-j)\}$$

This works since we want to minimise the score of the opposing player. So we pick the move that minimises his score.

This solution has a complexity of $O(NM)$ and will get you 50%. We can improve this by speeding up finding the minimum value at each point. We can modify the **best_score** function to be iterative, and then we can use a technique called the sliding window minimum.

The obvious sliding window minimum algorithm is just to keep track of the last M values and find the smallest in it. But that gives a final algorithm complexity of $O(NM)$. We can improve this by keeping a minimum heap which stores pairs of the form (value, index). Then when we want to get the current minimum, we pop all values whose index is out of range first, then use the remaining value as the minimum. This gives an algorithm of complexity $O(N\mathbf{log}(M))$ (which is the 80%).

Finding the minimum value can be done in asymptotically $O(1)$ time, yielding a solution which is $O(N)$ (which is the 100%). What we use is a deque, a list which supports quick ($O(1)$) removal and look-up of the back and front. The deque stores pairs of the form (value, index) and has the invariant

$$\mathbf{deque}[i].value < \mathbf{deque}[i+1].value \textbf{ and } \mathbf{deque}[i].index < \mathbf{deque}[i+1].index$$

for all $0 \leq i < \mathbf{deque}.size - 1$

If the deque has this invariant, then the smallest value is stored at $\mathbf{deque}[0].value$. If $\mathbf{deque}[0].index$ is outside of the window, you pop it from the front. Now to insert a value into the deque you remove all values from the back which cause the invariant to be false. Then you just push the value onto the back.